

# CREATING SCRIPTS AND AUTOMATED PROCEDURES

**After reading this chapter and completing the exercises, you will be able to:**

- ♦ Create shell scripts using basic shell programming features
- ♦ Automate one-time and repetitive tasks using `at` and `crontab`
- ♦ Manage automated system administration tasks

In the previous chapter you learned about configuring system log files to record information about what is happening in the Linux kernel and with other programs running on the Linux system. You also learned how you can use those log files to get information about system status and to enhance security.

In this chapter you will learn how to create shell scripts that automate tasks you would normally complete at a Linux command line. You will also learn how to automate system administration tasks via shell scripts that execute automatically at predefined times.

---

## WRITING SHELL SCRIPTS

As you have seen in previous chapters, working with Linux often involves entering a series of commands at the command line (using a shell such as `bash`). In many cases the commands you enter may be identical—or nearly so—to previously entered commands. You can automate the process of entering frequently used commands by creating a shell script. A **shell script** is an executable file containing lines of text as you would enter them at a command line, including special commands to control the order in which lines in the file are executed. Within a shell script you can store a series of commands that you would otherwise enter at a shell prompt. To execute the commands in a shell script, you simply execute the shell script.



A shell script is like a batch file in DOS or Windows. Batch files in DOS or Windows end with the file extension `.BAT`, for example, `AUTOEXEC.BAT`. The capabilities of shell scripts, however, go far beyond those of batch files.

Shell scripts can be used throughout a Linux system. The following list describes just a few examples of the shell scripts discussed in preceding chapters:

- The system initialization scripts in the `/etc/rc.d` subdirectory (such as `rc` and `rc.local`) are shell scripts.
- The scripts in `/etc/rc.d/init.d` that start system services such as a Web server are all shell scripts.
- The `/etc/profile` and `/etc/bashrc` scripts that Linux executes each time you log in are shell scripts.
- The scripts that the X Window System uses to launch initial graphical programs are shell scripts (for example, `/etc/X11/xinit/xinitrc` and `/etc/X11/xdm/xsession`).

Using shell scripts for all of these system functions has two great advantages. First, it allows you to study what is happening on the Linux system by reviewing the contents of the scripts that control various programs or system services. Second, you can change the way anything on the system occurs simply by altering the relevant shell script.

Before you continue reading this chapter, take time to locate some shell scripts on your Linux system (using the list above as a guide). As you learn about shell script features in this chapter, refer to these scripts to see how they are configured, what commands they include, and what each is designed to accomplish.

## Interpreted and Compiled Programs

Before you can create shell scripts, you need to be familiar with some basic programming concepts. First of all, you should be aware that there are two different types of computer programs: interpreted programs and compiled programs. Both types of programs are written using a computer language. A **computer language**, or **programming language**, is a set of words and syntax rules that can be arranged in predefined ways to cause a computer to perform tasks defined by the person using the language. The words used in a computer language are often called **keywords**, because they have special meanings when used within a computer program. For example, later you will learn the keyword `for`, which defines certain actions when used in a shell script.

A software developer or programmer writes a computer program using a computer language, storing the keywords and related information in a file. This file is considered the program's **source code**. The keywords that make up the computer language are human readable, though you must be familiar with the computer language to understand what the program will do. The computer cannot act directly on the keywords of the computer language. The keywords must first be converted to numeric codes that the computer uses. This numeric code

is called the **binary file**, or **executable file**. To run the program, the user must run the executable file.

The process of converting computer language keywords into computer-readable numeric codes can occur in two different ways:

- When the computer language is a compiled language, the source code is converted after the programmer writes the source code. The binary file is normally given to people who want to use the program. (Linux and related programs are unusual in that the source code is also available, but few people need to access it.) A **compiled language** is one for which the source code is converted to a binary file long before the program will be run by users. A **compiler** is a special program that converts the source code of a compiled language into a binary file.
- When the computer language is an **interpreted language**, the source code is converted into numeric codes at the instant a user runs the program. This conversion takes place each time the user runs the program. The source code is given to people who want to use the program. For this reason, interpreted languages are generally much slower than compiled languages. An **interpreter** converts the source code of an interpreted language into numeric codes.

A shell script is an example of an **interpreted program**, that is, a program written in an interpreted language. The commands that you learn in this chapter are part of the interpreted language for the `bash` shell. The shell is the interpreter that acts on the keywords that you include in shell scripts. All of these keywords can also be used at the shell prompt (at any command-line prompt). In general terms, a **script** is a text file that can be interpreted or executed by another program.

## Programming Concepts

Writing a shell script, though not overly difficult, is really computer programming. In order to write effective shell scripts, knowledge of a few programming concepts is helpful.

A computer program is executed one command at a time. Execution normally proceeds from the first line of the program to the last line. Each command within the program is also called a **statement**. The statement is often a single keyword, but the term *statement* may also refer to a group of keywords that the computer language syntax requires or allows to be used together. One such keyword, described later in this chapter, is `if`. You can never use the `if` keyword without also using the `then` keyword and the `fi` keyword (also discussed later in this chapter). This means that any references to an `if` statement actually refer to the `if` command and its associated information. Or you might hear of an `if/then` statement, which refers to the combination of several keywords that are normally (or always) used together. As you begin to learn about shell scripts, you will begin to think of the terms *command* and *statement* as being synonymous.

The real power of a computer program lies in its ability to decide which parts of the program to execute based on factors that the program itself examines. For example, if the program has the ability to check a stock price on a Web site, the program could be designed to perform one set of commands if the stock price is over a certain price, and another set of commands if the

price is below that price. This is done with a selection statement. A **selection statement** lets a computer programmer determine which parts of a program will be executed according to values that are calculated by testing as the program is executed. The use of a selection statement means that all lines in a computer program will not necessarily execute when the program is run—the results of the tests in the selection statements determine which steps run. Figure 12-1 illustrates a selection based on the results of a test.

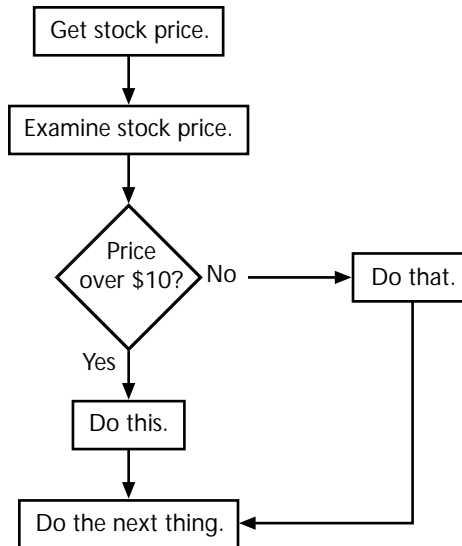


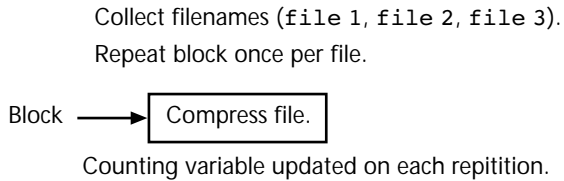
Figure 12-1 A selection statement choosing among alternative actions in a program

Most shell scripts include selection statements. They can be very complex, with many options, or very simple, with only a simple test such as “if this number is 4, do this; otherwise, don’t.” The tests performed in shell scripts (and in any computer programming language) only have two possible values: true or false. You must design your script to respond to these two possible values. Shell script tests are almost all related to checking the status of a file or the value of an environment variable. They are not designed for mathematical computations, though you can perform math functions using additional commands within a shell script.

A selection statement is used to determine whether part of a computer program should be executed. A **loop statement** is used to determine whether part of a computer program should be executed more than once. A loop statement has several parts:

- A counting variable, which is assigned a different value each time the loop is repeated. The counting variable is often called the **index** of the loop, because it tracks how many times the loop has executed a list of commands.
- A list of values that should be assigned to the counting variable.
- A list of commands that should be executed each time through the loop.

Figure 12-2 illustrates an example of a loop in which several files in a list are to be compressed. This example does not use shell script keywords, but only descriptive text to help you understand how a loop operates. Each of the files is named as part of the loop statement. The counting variable is assigned the value of a different filename each time through the loop.



**Figure 12-2** A loop statement repeats commands.

When using both selection statements and loop statements, you define a list of commands that are executed once, many times, or not at all depending on what happens when the selection or loop statement is executed. The list of commands (or statements) controlled by a selection or loop statement is often called a **block**, or a **statement block**. The statement block in Figure 12-2 is the single item, “Compress the file.” Statement blocks usually contain many statements. They can even contain other selection and loop statements. **Nesting** is a programming method in which one selection or loop statement contains another selection or loop statement.

## Components of a Shell Script

You can create a shell script in any text editor, including `vi`, `emacs`, or a graphical editor within KDE or Gnome. Any text file that adheres to these three basic rules is considered a shell script:

- The first line of the text file must indicate the name of the shell that should be used to execute the shell script.
- The text file must have the execute file permission set for the user, group, or other category (as you learned in Chapter 4).
- The text file must contain valid commands that could be executed at a command line.



A shell script is sometimes called a shell program. The process of writing a shell script is also called shell programming because it involves using a shell’s programming syntax.

When you execute a shell script, you are, in effect, launching a program. The program just happens to be contained in a human-readable text file rather than a standard binary file. But to the shell, both launching a shell script and launching a program such as WordPerfect require a similar process. In order for the shell to identify a text file as an executable script, the execute file permission must be set for the person attempting to execute the shell script.

For example, if you have created a text file in a text editor, the file permissions typically look like this (as viewed by the `ls -l` command):

```
rw-r--r--
```

To execute the shell script, you must activate the execute permission using this command. (Remember, you must be the file's owner or the `root` user to change file permissions.)

```
chmod ugo+x filename
```

You could also use the second form of the `chmod` command. This example grants a standard set of file permissions for a Linux program (using numeric codes for the `chmod` command):

```
chmod 755 filename
```

After using either of these `chmod` commands, the file permissions for the file look like this, indicating to the shell that the file can be executed as a program:

```
rwxr-xr-x
```

One difference between launching a shell script and launching a program like WordPerfect is that the shell must start another program to interpret (execute) the shell script. This is why the first line of the shell script must contain the name of the shell that will execute the script.

When a user executes a shell script (by entering the script name at a shell prompt), the shell examines the file to see what shell should execute the script. The shell then starts that shell and includes the shell script file as a parameter to be executed. If you are running the `bash` shell and launch a shell script that requires `bash`, the main shell starts a second copy of `bash` to execute the shell script.

The format of the first line of a shell script consists of a hash mark (`#`), followed by an exclamation point, followed by the shell name (with a complete path to the shell file). For example, the first line of a shell script written for the `bash` shell always looks like this, depending on the location of the `bash` shell program file within the Linux system (the location shown here is standard):

```
#!/bin/bash
```

Conversely, a shell script designed to be executed by the TC shell will have the following line as the first line in the script file:

```
#!/bin/tcsh
```

Many of the script examples in this chapter use the `echo` command. The `echo` command prints text (which you must enter after the `echo` command itself) to the `STDOUT` channel—to the screen unless output has been redirected. For example, to print the message “Hello world” to the screen, you would use the command `echo Hello world`. If the text after the `echo` command includes special characters, such as `(` or `*` you should enclose the text in quotation marks. Suppose you created a file named `testscript` containing the following lines (you could do this in any text editor):

```
#!/bin/bash
echo This is a sample shell script.
```



It doesn't matter what filename you use for your shell scripts. You can use any filename and any file extension, so long as the contents of the file itself adhere to the three rules listed previously.

To test a new script, enter the name of the file as a command at a Linux command line. Use a period and a backslash (`./`) before the filename to tell the shell that the file is located in the current directory rather than in a directory that is part of the `PATH` environment variable. For example:

```
./testscript
```

After you enter `./testscript` at the command line, the following steps will occur:

1. The shell you are working in looks at the first line of the `testscript` file and sees that it contains the line `#!/bin/bash`.
2. The shell you are working in launches a new `bash` shell with the filename `testscript` as a parameter. In effect, the shell executes this command for you:  

```
bash ./testscript
```
3. The new copy of `bash` loads the `testscript` file and executes each of the lines in the file as if they had been entered at a shell prompt, printing output from commands on the screen.
4. When the new copy of `bash` reaches the end of the `testscript` file, it exits, returning control of the screen to the shell from which the shell script was originally launched.

Figure 12-3 illustrates the execution of a shell script.

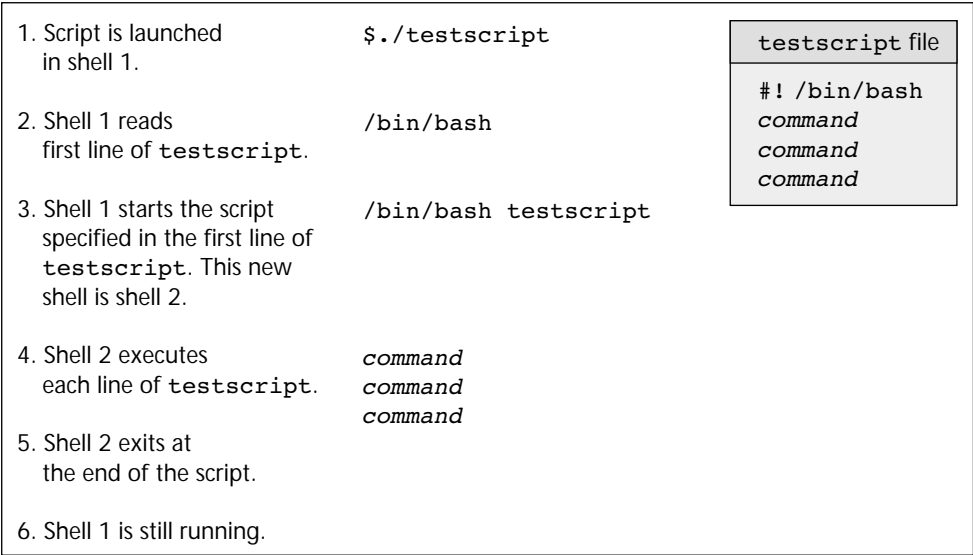


Figure 12-3 Execution of a shell script

The sections that follow describe how to create shell scripts based on the `bash` shell programming syntax. As you will recall from Chapter 6, other shells use different programming syntax rules. Because `bash` shell programming is much more common than other types of programming in Linux systems, the information provided here should serve you well. But you should always keep in mind that the shell you are working in and the shell used to execute a script can be different.

Suppose, for example, that you preferred to work in the `TC` shell because of its interactive shell features, but that you wanted to write a shell script that used `bash` programming syntax. To do this, you would start the shell script file with the `bash` shell name, using the format `#!/bin/bash` shown previously. As described in Step 2 above, the `TC` shell would launch a copy of the `bash` shell in order to execute the shell script. At the end of the shell script (after the last line in the file had been executed), the `bash` shell would exit, and you would again be working in the `TC` shell.

## Creating a Simple Shell Script

Some commands are generally used only within shell scripts. This section describes some of these basic commands and explains how you can use them to create simple shells scripts. For example, consider the following sample shell script:

```
#!/bin/bash
find /home -name core -exec rm {} \;
du /home >/tmp/home_sizes
```

The purpose of the second line of this script is to locate and remove all files named `core` within all users' home directories. In the third line, you use the `du` command to create a summary of the size of every subdirectory under `/home`, storing that information in a file named `home_sizes`. This script is not long, but the commands are somewhat complicated. By storing these lines in a script called `clean` (or any other name you select), you make it possible to execute both commands by entering this command:

```
./clean
```



The files named `core` that the preceding script example finds are produced when a Linux program ends unexpectedly. A developer can use the `core` file produced by a program to determine why a program ended. The `core` files can be quite large and are not needed unless someone is troubleshooting a recurring problem, so deleting them regularly can save a lot of hard disk space.

The `clean` script just shown doesn't produce any output on the screen. All of the commands in the script work directly with files. Other scripts write information to the screen or require input from the keyboard to process the commands in the script. The following script uses the `read` command. The `read` command causes the shell to pause for a user to enter

information at the keyboard. The information entered is assigned to a variable provided with the `read` command.

```
#!/bin/bash
echo Enter a filename to process:
read THEFILE
echo The number of lines in $THEFILE is:
wc -l $THEFILE
echo The number of words in $THEFILE is:
wc -w $THEFILE
echo End of processing for $THEFILE
```

This script uses a programming concept known as a variable. A variable is a memory location where a program can store a value that it needs in order to complete a task. You normally define a variable by assigning a value to a variable name that you choose. For example, including this line in a script defines a new variable named `MYFILE` and assigns it a value of `index.html`.

```
MYFILE="index.html"
```

Another way you can define a variable is to use the `read` command followed by a variable name. This command causes a script to pause for keyboard input. The text that a user enters at the keyboard is assigned as the value of the variable given in the `read` command. You can see an example of this in the previous script, where the third line contains `read THEFILE`. This command causes the script to pause, awaiting input. Whatever data is entered by the user running the script is assigned as the value of the variable called `THEFILE`.

Once a value is assigned to a variable using either of these methods (a direct assignment or using a `read` command), the script can use the variable within any other commands. In the example script just referred to, the script references the variable `THEFILE` as it executes the `wc` command.

If the above script were stored in a file called `filesize`, any user with execute permission to the file could launch the script using this command:

```
./filesize
```

The output from the `filesize` script is shown here. The script pauses after displaying the first line of this output so that the user can enter a filename. The filename shown (`report.txt`) would be entered by the user running the script.

```
Enter a filename to process:
report.txt
The number of lines in report.txt is:
453
The number of words in report.txt is:
3215
End of processing for report.txt
```

Although the `read` command is useful for collecting input from a user, you should make sure your scripts test the validity of values entered by the user. For example, if the user running the `filesize` script entered a filename that did not exist on the system, the other commands in

the script (the `wc` commands in this example) would generate error messages. Later in this chapter, you will learn how to create scripts that can test values entered by the user.

Because scripts use standard Linux commands that write information to the screen, they can also use the Linux redirection operators to change the flow of information to and from commands. As you learned in Chapter 7, you can use redirection operators to change how the standard input and standard output for any command are treated. For example, suppose you start the `filesize` script using this command:

```
./filesize > /tmp/output
```

Because of the `>` redirection operator, all of the data that would normally be written to the screen is instead written to the file `/tmp/output`. When you view the contents of the `/tmp/output` file using the following command, you see exactly what you saw on screen when you ran the `filesize` command without a redirection operator.

```
cat /tmp/output
```

You can use any of the Linux redirection operators (`<`, `>`, `>>`, or `|`) when you run any shell script. By using these operators, you can treat a shell script as you would treat any regular Linux command, using pipes to connect scripts with other commands, and so forth. You will see other examples of using redirection operators later in this chapter.

## Using Variables in Scripts

The `filesize` sample script shown in the previous section used a variable to store information entered by a user. A variable used in a shell script in this way is sometimes called a shell variable. Thus, a **shell variable** is a variable used within a shell script to store information for use by the script. You can work with shell variables and environment variables (which you learned about in previous chapters) in the same ways, though shell variables are usually defined by a person writing a new shell script, while environment variables are predefined by the Linux operating system scripts or programs running on Linux.

As you write shell scripts, you are likely to refer to many shell variables (which you define yourself by assigning a value within the shell script) and environment variables (which the shell defines by assigning them a value before executing the shell script). For example, a command in your shell script may need to copy a file into the home directory of the user running the shell script. To do this, you can refer to the `HOME` environment variable. The following sample command copies the file `report.txt` to a user's home directory:

```
cp /tmp/report.txt $HOME
```

Instead of the `HOME` environment variable, you could use the following command in the script, but this type of command only works if user `nwells` is running the script.

```
cp /tmp/report.txt /home/nwells/
```

By using the `HOME` environment variable, you ensure that the script works for any user who launches the script. As a system administrator, you should use techniques like this in order to create shell scripts that are as flexible as possible. This allows the scripts to be used safely by different system administrators and other users. Not all users can execute every

script, however. Some scripts access parts of the system that only the `root` user can access. If another user runs such a script, an error will occur.

Shell scripts often use special shell variables called positional variables. Rather than taking on a value assigned to it within the script, a **positional variable** takes on a value based on the information that the user includes on the command line (along with the command to run the script itself). If the `filesize` script shown earlier incorporated a positional variable, the user could enter the filename at the command line at the same time he or she executed the script. For example, the user would enter the following command to process the file `report.txt`:

```
./filesize report.txt
```

Within a script, positional variables are indicated using a dollar sign and a number. The notation `$0` indicates the first item on the command line (the script name, or in the example above, `./filesize`). The notation `$1` indicates the second item on the command line (in the example above, `report.txt`). The notation `$2` indicates the third item on the command line, and so forth. (Although the example above only includes two items, you can create scripts that refer to numerous items entered at the command line.) Thus, to incorporate a filename entered at the command line into a script, you would write the script as shown below. As you compare this script with the one shown in the section “Creating a Simple Shell Script,” you will see that in this script the `read` command is not used. Instead, the user executing the script must provide a filename on the command line. The shell will assign the filename to the variable `$1` as it starts the script. The `$1` positional variable is thus used in place of the `THEFILE` shell variable throughout the script.

```
#!/bin/bash
echo The number of lines in $1 is:
wc -l $1
echo The number of words in $1 is:
wc -w $1
echo End of processing for $1
```

Let’s assume the preceding version of the `filesize` script is executed using this command line:

```
./filesize myreport
```

The shell assigns the value of `myreport` to the variable `$1` before running the `filesize` script. The value of this variable is substituted in the script wherever you see a `$1`. So the commands that are executed when this script is run (as shown above) look like this:

```
echo The number of lines in myreport is:
wc -l myreport
echo The number of words in myreport is:
wc -w myreport
echo End of processing for myreport
```

Depending on the size of the `myreport` file, the output of the script might look something like this:

```
The number of lines in myreport is:
452
The number of words in myreport is:
3419
End of processing for myreport
```



As you've probably guessed, the term *positional variable* refers to the fact that the position of the items on the command line dictates the variable names to which they are assigned within the script.

As you work with positional variables, it's often helpful to know how many items the user running the script has included on the command line. For example, you may want to include commands to verify that the correct number of items is included on the command line before having the script proceed with execution. Each time you execute a script, the shell defines a special variable called `$#` that contains the number of items on the command line used to execute the script. Later you will learn about commands that enable your scripts to test the value of the `$#` variable to determine how many items a user provided on the command line when the script was launched. For example, suppose you created the following script to output information about several filenames entered at the command line:

```
#!/bin/bash
echo The script you are running is $0
echo The number of parameters you provided to process is $#
echo The number of lines in file $1 is:
wc -l $1
echo The number of lines in file $2 is:
wc -l $2
echo The number of lines in file $3 is:
wc -l $3
echo The number of lines in file $4 is:
wc -l $4
```

If the above script were stored in a file named `info`, you would launch it like this, with the names of four data files included on the command line:

```
./info data1 data2 data3 data4
```

The shell assigns the name of the script—the first item on the command line—to the positional variable `$0`. The shell also assigns the filenames included on the command line to the positional variables `$1`, `$2`, `$3`, and `$4`, respectively. Finally, the shell assigns the value of 4 to the `$#` variable (which you see on the third line of the script) because the command line used to execute this script contains four items besides the name of the script. Thus the commands executed when this script is run are as follows:

```

echo The script you are running is info
echo The number of parameters you provided to process is 4
echo The number of lines in file data1 is:
wc -l data1
echo The number of lines in file data2 is:
wc -l data2
echo The number of lines in file data3 is:
wc -l data3
echo The number of lines in file data4 is:
wc -l data4

```

The output of the script would then look like this (depending on the size of the data files):

```

The script you are running is info
The number of parameters you provided to process is 4
The number of lines in file data1 is:
123
The number of lines in file data2 is:
11241
The number of lines in file data3 is:
2321
The number of lines in file data4 is:
3159

```

Positional variables are a very useful way to provide information to the commands in a script. But a script such as this example requires that a user provide a precise number of filenames; if the user makes a mistake, the script will generate error messages. In the next section you learn how to test values in a shell script and act on the result of those tests.

## Understanding Tests

In many scripts it is useful to determine whether a given condition is true or false. The following list provides a few examples of when such a test is useful.

- You need to determine whether a filename (that is entered by a user or included on the command line) actually exists.
- You need to see if two numeric values are identical.
- You need to determine whether a file is empty.
- You need to test whether the number of parameters on the command line is correct, thus allowing the script to function as intended.

A **test** is a method of examining something within a shell script such as a file or a variable. The script chooses which commands to execute according to the result of the examination (that is, according to the result of the test). For example, a script might test whether a filename referred to a valid file, or whether a certain variable value was too low for the needs of the script. The **if** command is used to introduce a test within a shell script. An **if** command is always followed by a **then** command. A **then** command identifies the other commands that should be executed if the test introduced by the **if** command succeeds (returns a value of true). The **fi** command marks the end of a set of commands that begin with an **if/then**

command pair. If the test within an `if` command succeeds, all of the commands between `then` and `fi` are executed. Otherwise, none of those commands are executed. The `test` command evaluates the parameters that you include in the script after the `test` command and returns either true (a value of 1) or false (a value of 0). The results of the `test` command are then evaluated by the `if` command. An example of an `if` command used with a `test` command is shown here. (The parameters of the `test` command are described later.)

```
if test $1 -eq report
```

Instead of the `test` command shown in this example, square brackets are commonly used around the parameters that you would use with the `test` command. The shell uses the `test` command to evaluate the parameters in square brackets, and the result is used by the `if` command.

Figure 12-4 shows the format of the `if` command used with square brackets to test parameters as just described. This arrangement is called an `if/then` statement or an `if/then` block.

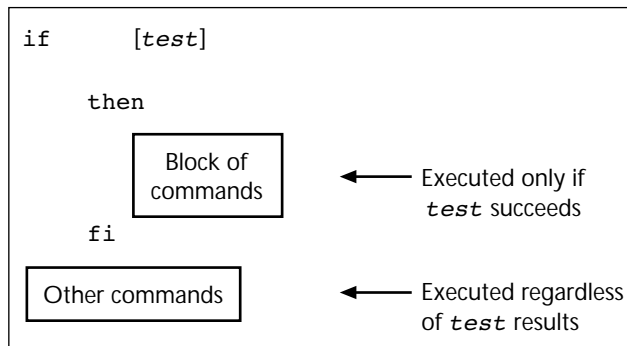


Figure 12-4 Structure of an `if/then` statement

To form a test in the `bash` shell, you include one or two items being tested along with a test operator that defines how you want to test the items. The items being tested and the test operator are like command parameters. All of the parameters are enclosed between square brackets. Table 12-1 shows the testing operators that are available in the `bash` shell.

## Using `if/then/else` Statements

An **`if/then/else` statement** is another kind of selection statement. Such a statement specifies that *if* a test returns a value of true, *then* one set of commands should be executed; otherwise (*else*) another set of commands should be executed. Such a statement begins with an `if` command, such as the one shown below. In this example, the test being performed (that is, the item between the square brackets) could be described like this: “if the file `/etc/smb.conf` exists, return a value of true; otherwise, return a value of false.”

```
if [ -f /etc/smb.conf ]
```

Table 12-1 File-Testing Operators in the `bash` Shell

Test operator	Description
<code>-d</code>	Test whether the item is a directory name.
<code>-e</code>	Test whether the filename exists.
<code>-f</code>	Test whether the file exists and is a regular file.
<code>-h</code> or <code>-L</code>	Test whether the file exists and is a symbolic link.
<code>-r</code>	Test whether the file exists and is readable
<code>-w</code>	Test whether the file exists and is writable
<code>-x</code>	Test whether the file exists and is executable.
<code>-lt</code>	Test whether the numeric variable on the left (of the operator) is less than the numeric variable or value on the right.
<code>-gt</code>	Test whether the numeric variable on the left (of the operator) is greater than the numeric variable or value on the right.
<code>-le</code>	Test whether the numeric variable on the left (of the operator) is less than or equal to the numeric variable or value on the right.
<code>-ge</code>	Test whether the numeric variable on the left (of the operator) is greater than or equal to the numeric variable or value on the right.
<code>-eq</code>	Test whether the numeric variable on the left (of the operator) is equal to the numeric variable or value on the right.
<code>-ne</code>	Return a value of true if the numeric variable on the left (of the operator) is not equal to the numeric variable or value on the right.

After this `if` command, you can add a block of commands that you want executed only if the test returns a value of true. (This is the *then* part of an `if/then/else` statement.) As a simple example, consider this partial script:

```
if [ -f /etc/smb.conf ]
then
    echo The Samba server appears to be configured.
fi
echo Processing the server configuration.
```

In the above example, the `-f` test operator used in the `if` command means that the `echo` command is only executed when the `if` command returns a value of true—that is, if the file `/etc/smb.conf` does indeed exist. If the file does *not* exist, the shell skips the statements between the `if` and `fi` commands, and proceeds to execute the next command after the `fi` command. In the above example, the next command would be the `echo` command with a message stating that the server configuration is being processed.



The indentation of lines using tabs and spaces is a convention designed to make the script easier for a person to read. Tabs and spaces included in a program are called **white space**. White space between commands or lines has no effect on a shell script's operation.

The **else** command extends the capability of an **if/then** statement by adding a block of commands that are only executed if a test returns a value of false (that is, if the test fails). You can use an **else** command only after an **if/then** command. The structure of an **if/then/else** statement is shown in Figure 12-5. Notice especially that the **else** block of commands is skipped if the test returns a value of true.

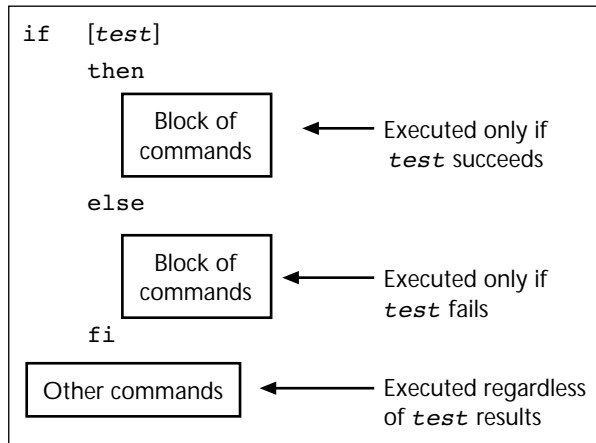


Figure 12-5 Structure of an **if/then/else** statement

The following sample script shows how the **else** command is used. As with the previous example, the **fi** command ends the entire statement. The commands between **then** and **else** are executed only if the test succeeds. The commands between **else** and **fi** are executed if the test fails.

```

if [ -f /etc/smb.conf ]
then
    echo The Samba server appears to be configured.
else
    echo The Samba server cannot be started.
fi

```

The **exit** command stops the execution of a script immediately. The **exit** command is often included within an **if/then/else** statement, and it is then executed only if certain conditions exist (as determined by an **if** command).



A **comment** is a line within a script that is not processed by the shell, but is only included to help someone reading the file understand the purpose of the script or how it functions. To include a comment, begin a line with a hash mark, **#**. (The first line of a script that begins with **#!** is a special case and is not a comment.)

The following script includes a test in which the number of parameters included on the command line that launched the script is tested within an **if/then** statement. Specifically,

the test determines whether the \$# variable has a value of 2 or not, using the `-ne` operator to test for “not equal to.” Notice that the script includes several comment lines. You should include numerous comments to help other system administrators understand the scripts you write (and to remind yourself how and why you created a script when you review it months later).

```
#!/bin/bash
#
# Script to add a user to the database user's list
# Requires a username and database table name on the command line
# launching the script
# Created 20 April 2000; Nicholas Wells
#
if [ $# -ne 2 ]
then
    echo You must provide a username and a database table name.
    exit
fi
# Begin processing: Store positional variables in new shell variables
DB_USER=$1
DB_TABLE=$2
```

Because Linux systems use many environment variables and complex test expressions involving multiple comparisons, you may have trouble understanding all of the shell scripts on your Linux system until you have reviewed the test syntax thoroughly and seen many examples of complex `if/then` statements. The `bash` shell supports several additional commands to extend the capability of tests. Those additional testing commands are beyond the scope of this book.

## Adding Loops to a Script

Recall that a loop is a block of commands that can be executed more than one time without the user having to restart a shell script. A loop can repeat a block of commands a certain number of times or indefinitely until a condition is met (that is, until a test succeeds). The `bash` shell supports several commands that you can use to define a loop within a shell script.

The `for` command repeats a block of commands one time for each item in a list. The block of commands that is repeated is known as a **for loop**. You can include a list of items when you write the script, or you can design the loop so that the list items are provided on the command line when the script is executed. The syntax of a `for` command is shown below. The parameters that you must include after the `for` command are a counting variable and a list with each value that you want the counting variable to take. The commands between the `do` command and the `done` command are executed one time for each item in the list. The shell assigns a new value to the variable each time the loop executes—specifically, it assigns each value in the list in turn. The `do` and `done` commands are keywords used to begin and end a block of commands to be executed by a `for` command.

```
for variable in <list of items>
do
    block of commands
done
```

The list of items in a `for` command can be any of the following:

- Numbers
- Words
- A regular expression used to match filenames in the current working directory
- The special variable `$@`, described below

Each of these possible methods of defining a list of items in a `for` loop is shown in the following example script lines:

- `for COUNT in 1 2 3 4 5 6 7`
- `for NAME in george ali maria rupert kim`
- `for db_filename in *c`
- `for db_filename in $@`

When you use a regular expression (like `*c`) as the list of items in a `for` command, the shell replaces the regular expression with all matching names in the current directory. Using the example of `*c`, the shell will insert all the filenames in the current directory that end with the letter `c`. This occurs before the shell executes the `for` command itself.

The `$@` variable used in the last example above is a special variable that includes all of the parameters on the command line. This variable is often used when you create a script that processes any number of filenames given on the command line. The following example script shows how a loop processes each filename on the command line in turn:

```
#!/bin/bash
for i in $@
do
    gzip $i
done
```

Suppose the above script were stored in a file called `convert`. If you started the script using the following command line, the `for` loop would execute the commands between `do` and `done` (the `gzip` command in this example) three times, once for each filename on the command line.

```
./convert phoebe.bmp charon.bmp europa.bmp
```

Figure 12-6 shows how the value assigned to the `i` counting variable changes during each of the three times through the script. As you can see, Figure 12-6 uses the term *iteration*. Each time through the loop counts as one **iteration**. Programmers sometimes use the phrase “iterating through a loop,” meaning that the block of commands within the loop is repeated, with new variable values assigned each time through.

for statement:	for i in \$@
\$@ replaced with file names:	for i in phoebe.bmp charon.bmp europa.bmp
Value of i:	First iteration, i=phoebe.bmp Second iteration, i=charon.bmp Third iteration, i=europa.bmp

Figure 12-6 Iterations through a simple `for` loop

A second type of loop used in shell scripts does not include a list of items to use as variable values. Instead, this second type of loop includes a test like an `if/then` statement. As long as the test returns a value of true, the block of commands in the loop is executed again and again. As soon as the condition changes (the test returns a value of false), the loop exits. You use the `while` command to create such a loop. After a `while` command, you include a block of commands to execute as long as the test returns true. The syntax of the `while` command is shown here. It is similar to the `for` loop in that the block of commands constituting the loop is positioned between `do` and `done`.

```
while <test>
do
    block of commands
done
```

You can perform the same tests with a `while` command that you can with the `if` and `then` commands. For example, the following partial script requests a filename from a user. The loop repeats, asking for the filename again, until the filename entered is a valid file. The test in this example is a combination of the `-f` operator (which tests whether the file exists) and the `!` operator (which reverses the result of the test). The `!` operator is used because this script is designed to execute a block of commands if something is not true. Rather than have a separate set of operators such as “not a file,” and “not a directory,” you simply use a `!` operator with one of the other operators. You could read the test shown in the `while` command as “while not `DB_FILE` exists”—in other words, while the file `DB_FILE` does not exist—keep looping through the following commands (to try to collect a valid filename from the user). Notice that the programmer has assigned a value to the variable used within the test portion of the `while` statement *before* the `while` statement is executed. This lets the programmer be certain of the variable value (and thus the test result) the first time that variable is tested.

```
DB_FILE=" "
while [ ! -f $DB_FILE ]
do
    echo Please enter the database file to archive:
    read DB_FILE
done
```

The script will repeat the block of commands between `do` and `done` indefinitely unless a valid filename is entered. Note, however, that the user can generally exit any script by pressing Ctrl+C.

## Other Scripting Methods

So far, you have learned about creating shell scripts, but many other types of scripts are also commonly used in Linux. The commands in a shell script must follow specific syntax rules for tests, loops, and so forth. Other types of scripts—which are written using different computer languages and which are executed by different interpreters (rather than the `bash` shell)—require you to follow different syntax. The specific syntax rules depend on the interpreter that will execute the script. Each interpreter (such as Perl or Python) enforces its own set of syntax rules to communicate a specific set of instructions to the computer.

Different types of scripts are used for different purposes. After you learn about the types of scripts available (or more precisely, about the programming languages that you can use to create scripts), you will be able to decide which type of script is best suited to helping you complete a given task. In every case, the three rules regarding shell scripts given at the beginning of the section “Components of a Shell Script” still apply. That list is shown again here.

- The first line of each script file identifies the interpreter that will execute the script.
- The script file must have the execute file permission set.
- The script file must contain commands that are valid for the interpreter that will execute the script.

Table 12-2 lists several popular scripting languages used in Linux. All of these languages are included in most standard Linux distributions. Many books are available to teach you how to write programs using any one of these scripting languages.

Table 12-2 Popular Scripting Languages

Language	Comments
Perl	Used extensively to process data on Web servers. Rarely used for graphical programs. Very popular and well known.
Awk	Used for system administration work, often processing text files in conjunction with the <code>sed</code> program (see Chapter 6). The most widely used version of the <code>awk</code> interpreter used on Linux is the <code>gawk</code> program from the GNU project.
Python	A fairly new scripting language that is rapidly gaining popularity. Excellent for creating graphical programs.
Tcl/Tk	A very popular language for developing graphical applications quickly.
Expect	Used by system administrators to interact with programs that normally require keyboard input.

As you have already learned, file extensions are not always used in Linux. Binary Linux programs such as `ls` and `cp` don't even include a file extension (like the `.EXE` or `.COM` used on Windows systems). Scripting languages are no different. By convention, however, script files often use standard file extensions to help users identify the information in a file. For example, Perl scripts often end in `.pl`, Python scripts in `.py`, and Tcl/Tk scripts in `.tcl`. You'll even see shell scripts that end with an `.sh` file extension. Just remember that these are conveniences for the user of the system, not requirements of the programs themselves. The next two sections describe some of these scripting languages in detail.

## Perl Scripts and CGI

Working with Perl scripts is similar to working with shell scripts in many ways. One of the most widely used scripting languages is Perl. **Perl** is a programming language developed by Larry Wall initially to process strings of text and generate reports. Eventually Perl became very popular for creating powerful scripts that control the interaction between the Web server and a user running a Web browser.

When you launch a script by entering its filename at a shell prompt, the shell examines the first line of the script to determine which interpreter will execute the script file. The first line of a Perl script file would therefore look like this (depending on where the Perl interpreter is located on the Linux system):

```
#!/usr/bin/perl
```



The name Perl is actually an acronym for Practical Extraction and Reporting Language, but you will rarely hear this longer term used.

Web servers are designed to use the standards of the **Common Gateway Interface**, or **CGI**, which is a method of communication between two programs, such as a Web server and a Perl script. CGI employs the standard input and standard output channels to facilitate communications. Environment variables and other tools are also used to facilitate this communication. CGI is especially useful in situations where a user needs to submit information to a Web server by means of a form on a Web page. To understand the usefulness of CGI, consider the following scenario:

1. A user running a Web browser enters data (such as the user's name and e-mail address) in a form.
2. The browser sends the data from the form to a Web server running on Linux.
3. The Web server starts a Perl script designed to process the data in the form.
4. The Perl script retrieves the form data submitted by the browser and acts on it (perhaps adding it to a database).
5. The Perl script creates a customized response for the user based on the data that the user entered and writes that response text to standard output.

6. The Web server collects all the output of the Perl script and sends it back to the browser as a document.

The interaction between the Perl script and the Web server in Steps 5 and 6 unfolds according to the standards of CGI. By using CGI standards, the Web server can rely on standard communication channels as a type of gateway between two programs. To achieve higher performance, several other methods (such as using Active Server Pages or embedding a Perl interpreter within a Web server) are now commonly used for exchanging data between Web servers and customized programs. But CGI is still an important tool and one that a Linux administrator must understand. You need to be familiar with the way Linux programs (in particular Linux network services) incorporate the methods of CGI—that is, how these programs gather data from other programs using the standard Linux communication channels.

## Scripts for Graphical Programs

You can use other types of scripting languages to create graphical programs that include dialog boxes and menus. A graphical program written in a scripting (interpreted) language performs much slower than one written in a compiled language (such as C or C++). This is because the interpreter must prepare everything for the graphical display the moment the program is run. By contrast, in a compiled program the many steps required to display a graphical interface are handled before the program is actually started. Using scripts to create graphical programs is often useful, nevertheless, because it normally takes much less time to write a script than to create a C or a C++ program.

Two of the most popular scripting languages for creating graphical programs are Tcl/Tk and Python. **Tcl/Tk** is a scripting language developed by John Ousterhout. The name Tcl/Tk stands for *tool control language/toolkit*. The word *toolkit* refers to the graphical toolkit, or set of programming functions that a programmer can use within Tcl/Tk scripts for creating graphical interfaces. (Note that Tcl/Tk is often referred to as “tickle-TK.”)

Scripts written in Tcl/Tk rely on an interpreter called `wish`. Thus the first line of a Tcl/Tk script includes the name `wish` rather than the name Tcl or Tk. For example, the first line of a Tcl/Tk script might look like this:

```
#!/usr/bin/wish
```

The scripting language Python uses the same Tk graphical programming toolkit as Tcl. **Python** is an object-oriented programming language, meaning that Python programs are designed so that parts of one program can be reused in another program with minimal effort. Python was developed by Guido van Rossum. You can use either Tcl/Tk or Python to create powerful and complex graphical programs. Figure 12-7 shows an example of a graphical program written in Python.

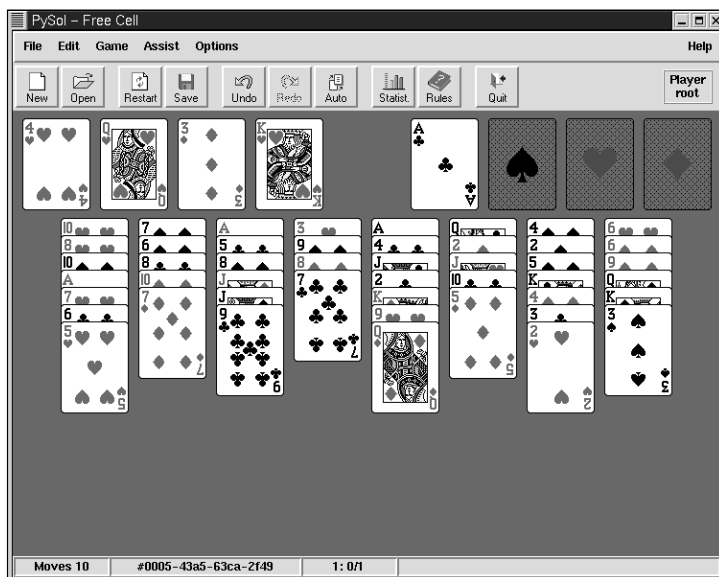


Figure 12-7 Sample graphical program written in Python

## AUTOMATING TASKS WITH `at` AND `crontab`

Scripts are not the only means for automating a series of commands in Linux. This part of the chapter describes some other tools that let you define a task and have it execute automatically, either when you are doing other work on the system or when you are away from the system (such as in the middle of the night).

The `at` command lets you enter one or more commands that will be executed once at some future time. The `crontab` command lets you enter one or more commands that will be executed repeatedly at intervals that you designate. Both commands are included in all standard Linux distributions. The `at` command relies on a daemon called `atd`; the `crontab` command relies on a daemon called `crond`. Both of these daemons are started by the standard system initialization scripts located in `/etc/rc.d/init.d`. Each of these programs (`atd` and `crond`) checks once per minute for any scheduled tasks and then executes them.

A scheduled task is often called a job, and you will often hear of cron jobs. A **cron job** is a command or script that you have scheduled to be executed at a specific time. The term cron job is sometimes used to refer to tasks you have set up using the `at` command as well. Some scheduled tasks, or cron jobs, only need to be performed once. Others are repetitive. The next section describes how to use the `at` command to schedule cron jobs (or *at* jobs, as they are sometimes called) that only need to be executed one time.

## Automating One-Time Tasks

Sometimes you may want to arrange for a single task to be executed once at some point in the future. The following list describes a few cases when this might be true:

- You need to start a back-up operation after all employees have left for the evening.
- You want to start a large database query during lunch, but you need to leave the office early.
- You need to remind several users on the network of a meeting occurring at 3:00 p.m., but you're afraid you'll forget it yourself as well.

In each of these cases, you can use the `at` command, as described in the next section.

### The `at` command

To automate a task with the `at` command, you can either enter the commands you want to schedule for later execution directly at the command line, or you can include them in a file. For long lists of commands, it's best to include them in a file so you can check the accuracy of the commands before using the `at` command.

If you have created a text file containing the commands you want to enter, the format of the `at` command looks like this:

```
at -f filename <time specification>
```

For example, suppose you have stored the following `tar` command in a file called `archive_job`. (As described in Chapter 14, the `tar` command is used to back up a Linux system.)

```
tar cvzf /archive/home_dirs.tgz /home
```

Suppose further that you want to start the archive operation at 11:30 p.m. The following `at` command would execute the command for you:

```
at -f archive_job 23:30
```

As described previously, the `atd` daemon will check once per minute for any jobs that have been scheduled using the `at` command. At 11:30, the job scheduled by the above command will be executed. Any results from the command (that is, text that would normally be written to the screen) will be e-mailed to the user who entered the `at` command. If you prefer to have the output stored in a file, you rerun the `at` command after altering the `archive_job` file as follows:

```
tar cvzf /archive/home_dirs.tgz /home > /tmp/archive_listing
```

The time specification in the `at` command is very flexible. You can include a standard hour format, such as 23:30 in the previous example, but you can also include items such as the words `now`, `minutes`, `hours`, `days`, or `months`. You can include the words `today`, `tomorrow`, `noon`, `midnight`, and even `teatime` (4:00 p.m.). Table 12-3 shows several examples of specifying a time with the `at` command.

Table 12-3 Time Specifications in the `at` Command

Time specification	Description
<code>at -f file now + 5 minutes</code>	Execute the commands in <code>file</code> five minutes after the <code>at</code> command is entered.
<code>at -f file 4pm + 5 days</code>	Execute the commands in <code>file</code> at 4:00 p.m., five days from now.
<code>at -f file noon Jul 31</code>	Execute the commands in <code>file</code> at noon on July 31.
<code>at -f file 10am 08/15/00</code>	Execute the commands in <code>file</code> at 10:00 a.m. on 15 August 2000. (The <code>at</code> command has no problems with Year 2000+ dates.)
<code>at -f file 5:15 tomorrow</code>	Execute the commands in <code>file</code> at 5:15 a.m. tomorrow morning.

Note the following points about specifying times within the `at` command:

- Each item in the time specification is separated by a space.
- You cannot combine multiple phrases (such as 4 hours and 25 minutes).
- When you use the word `now`, it should be the first word of the time specification.
- When a time is specified (such as 23:30), the command file will be processed the next time the system clock reaches the indicated time. For example, if it is currently 9:00 a.m. and you want to schedule a task for 11:00 a.m. tomorrow, you must indicate `tomorrow` in the time specification or the task will be executed at 11:00 a.m. today (because 11:00 a.m. has not passed yet today).
- If the `at` command cannot understand the time specification, you see a message stating `Garbled time`, and the command is not accepted for processing.

When you enter an `at` command with both a valid file that contains commands and a valid time specification, you see a message like the following. (The job number and the time displayed depend on the state of the Linux system.) This message reminds you that the commands you enter will be executed using the `/bin/sh` program. (On most Linux systems this is another name for the `bash` shell.) It also indicates the exact date and time when the command file will be executed.

```
warning: commands will be executed using /bin/sh
```

```
job 9 at 2000-10-21 05:15
```



The filename that you include as a parameter to the `at` command can include a shell script that you have written. If you have already prepared a shell script for some task, referring to that shell script in an `at` command makes it easy to schedule a time to launch the script. In addition, you can create a shell script that relies on another shell (besides the `/bin/sh` shell used by default).



When you create a command file for the `at` command, you don't need to begin it with the line `#!/bin/bash`, as you would with a script. Commands in the file are executed by the `/bin/sh` shell as if you had executed them at the shell prompt.

## Using `at` Interactively

The format of the `at` command shown in the preceding section requires you to prepare a file containing the commands that you want to execute. If you prefer, you can enter commands at the shell prompt so that you don't have to prepare a command file before using the `at` command. The disadvantage to entering commands interactively is that you cannot edit the commands after you enter each line; the advantage is that you don't need to create a separate file before scheduling a task.

To use the `at` command interactively, simply omit the `-f filename` portion of the `at` syntax. For example, you might enter a command such as this:

```
at now + 5 hours
```

When you press Enter after typing such an `at` command, you see a prompt that looks like this: `at>`. At this prompt you enter the command that you want the `atd` daemon to execute. You can enter multiple lines of text. You can also enter shell programming commands. For example, you can enter `if/then` statements or `for` loops, as described earlier in the chapter. Suppose, for instance, that you entered a `for` loop that uses the `mail` command to e-mail a short list of text files. The following would appear on your screen after you had entered all of the lines of the `for` loop. (Press Enter after each line shown here.)

```
at now + 5 hours
at> for file in *.txt
at> do
at> mail -s "File $file" tomr < $file
at> done
at>
```

After entering all the commands that you want the `atd` daemon to execute, you must indicate that you have finished entering commands. To do this, you press the `Ctrl+D` key combination, which sends an end-of-input character to the `at` command. At this point, the text `<EOT>` is displayed on the last line containing the `at>` prompt, followed by a message indicating the job number and time that the `atd` daemon will execute the command. Then the shell prompt returns.

When you enter a series of commands at the `at>` prompt, the results of the commands are e-mailed to you, just as they are when you specify a command file using the `-f` parameter. The e-mail message that the `atd` daemon sends you has a subject such as "Output from your job 11," where the number at the end of the message (11 in this example) is the job number assigned when you entered the `at` command.



For the specified task to be completed, the Linux system must be turned on at the time you specify in the `at` command. For example, if you indicate that a command should be executed in four hours, but the computer is not turned on in four hours, the command will not be executed then, nor will it be executed when the system is turned on again. The `atd` daemon does not check for commands that could not be executed because the computer was turned off.

Sometimes you'll want the output from a command to appear on screen instead of being e-mailed to you. For example, if you need to send yourself a reminder, having it sent to e-mail won't be much help. If you remember to check for the e-mail message, you'll remember the event you were to be reminded of. As an alternative, you can use the `tty` command to send output from a command (previously scheduled with `at`) to the location where you are currently logged in. The `tty` command returns the name of the terminal device that you are currently working in. For example, if you have logged in on the first virtual console of a Linux system and you enter the command `tty`, you see the device name `/dev/tty1` displayed on screen. You can use the output of this command to redirect output of a command that you submit to the `at` command. For example, suppose you submitted the following command to the `at` command (either using a separate command file or directly at the `at>` prompt). Notice that the `tty` command at the end of this command is enclosed in single backward quotation marks:

```
echo Go to your 401k meeting in conference room 6! > `tty`
```

When the `atd` daemon processes this line, the `tty` command is executed first, and the output from the `tty` command (such as `/dev/tty1`) is substituted for the ``tty`` text. The `echo` command then sends the line of text to the terminal where you are working (such as `/dev/tty1`). Any error messages will still be sent to standard output (and thus e-mailed to you). You can also redirect the Standard Error channel using this format:

```
echo Go to your 401k meeting in conference room 6! 2>&1 > `tty`
```

The Standard Error channel is number 2. The above statement redirects channel 2 to channel 1, which is standard output. Standard output is then redirected to the device name output by the `tty` command, which will be the device where you are logged in.



If you are working in multiple graphical terminal windows, the `tty` command may not allow the `atd` daemon to send the output to the precise window in which you are working. Don't rely on this method for sending critical, time-sensitive reminders to yourself.

## Using the `batch` Command

A command similar to the `at` command, and which also relies on the `atd` daemon, is the `batch` command. Rather than running commands at a specified time, the `batch` command runs your commands when the system load average drops below 0.8. (You can see the system load by using the `top` command.) In addition, if you enter multiple commands at the same time using the `batch` command, to avoid overloading the system, the `batch` command only starts one command at a time. The `batch` command is useful for times when you want to run a CPU-intensive program but the system is currently very busy with other

tasks. In this case, you can use the `batch` command to enter the name of the program you want to run. As soon as the system is no longer so busy, the command will be executed. As with the `at` command, the results of commands run by `batch` are e-mailed to you.

For example, suppose you needed to start a time-consuming task such as reconfiguring the Linux kernel using the `make` command. You could enter the `batch` command as shown below. Notice that no time parameters are included; the commands run as soon as the system load permits. You can include the `-f` option with the `batch` command, or just use the command name and then enter commands at the `at>` prompt, as you would when using the `at` command.

```
batch
at> cd /usr/src/linux
at> make dep; make zImage
at><EOT>
```

As with the `at` command, you press Ctrl+D when you have finished entering the commands you want `batch` to process. The `<EOT>` designation then appears, and the command prompt returns. As soon as the system load was small enough, the `batch` command would execute the commands you entered.



The **make** command in the preceding example uses commands stored in a configuration file to compile source code into a binary program, or otherwise prepare a program to be executed. This command uses a file called `Makefile` (which acts like a control script) to determine what actions to take when you enter a command such as `make dep` or `make zImage`. To learn more about `make`, enter the command `man make`.

## Automating Recurring Tasks

You can use the `crontab` command to prepare tasks that you want to have executed automatically at regular intervals. You might use `crontab` when you need to:

- Create a list of files that have not been accessed in a long time to check whether they can be deleted or archived
- Create a back-up copy of all active files (those recently accessed)
- Compile a list of all directories on the system, sorted by size, to help you identify areas that are using a lot of hard disk space
- Remove core files or other unused files that are using a lot of hard disk space
- Delete files in the `/tmp` directory that have not been used recently
- Rotate log files to keep them from becoming too large
- Run security scanning software (for example, check e-mail attachments for viruses or search system log files for multiple login failures)
- Store the results of the `ps` or `df` command to make a snapshot of the system's state at different times

Many Linux distributions include a simple method of automating tasks that doesn't require you to use the `crontab` command. In Red Hat Linux, the `/etc` directory contains subdirectories named `cron.hourly`, `cron.daily`, `cron.weekly`, and `cron.monthly`. You can place a script in any of these subdirectories. That script will be executed hourly, daily, weekly, or monthly, depending on which directory you place the script in. This method does not let you specify a precise time for your script, but most system administration tasks don't require a precise execution time; the important point is not to run them during the work day when most systems would be busy with many other tasks.

Every Linux system provides an `/etc/crontab` file, which shows the format of a standard entry for the `crontab` command. The `/etc/crontab` file in Red Hat Linux is shown here. This example file uses a special script from Red Hat Linux called `run-parts`.

```
SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root
HOME=/

# run-parts
01 * * * * root run-parts /etc/cron.hourly
02 4 * * * root run-parts /etc/cron.daily
22 4 * * 0 root run-parts /etc/cron.weekly
42 4 1 * * root run-parts /etc/cron.monthly
```

Notice the following about the example `/etc/crontab` file above:

- The `SHELL` variable defines which shell the `crond` daemon will use to execute the commands and scripts listed in the file.
- The `MAILTO` variable defines which user on the Linux system will receive an e-mail message containing the output from all cron jobs defined in the file.
- The `HOME` and `PATH` variables define a working directory and the directories where system commands are located. These variables will be used by all commands and scripts used in this file (and hence by all scripts in the `cron.daily` through `cron.monthly` subdirectories).
- The lines that end with the names of the subdirectories (such as `cron.daily`) include the username `root`. All of the cron jobs that you place in the `cron.daily` and related directories will be executed via the `root` user account.

The flexibility provided by the `crontab` command makes it challenging to use. For example, the preceding sample file includes numbers and asterisks that define when the commands and scripts will be executed. To use `crontab` effectively, you must learn well the format of the fields containing the numbers and asterisks. Each `crontab` specification begins with five fields, described in the following list as they appear on each `crontab` line from left to right:

- *Minute of the hour*: this field can range from 0 to 59.
- *Hour of the day*: this field can range from 0 to 23.

- *Day of the month*: this field can range from 0 to 31, but be careful about using the days 29, 30, or 31, because then nothing will happen in the month of February.
- *Month of the year*: this field can range from 0 to 12. You can also use the first three letters of a month's name (in upper- or lowercase).
- *Day of the week*: this field can range from 0 to 7 (0 and 7 are both Sunday; 1 is Monday). You can also use the first three letters of a day's name (in upper- or lowercase).

The `crond` daemon examines all five of these fields once per minute. If all five fields match the current time, `crond` executes the command on that line. When you use an asterisk in a field, `crond` ignores that field. In effect, placing an asterisk in a field means “execute this command no matter what the current value of this field is.” For example, including an asterisk in the day of the month field means “execute this command no matter what day of the month it is.” In order for the command to be executed, all five fields must match the current time and date. By placing an asterisk in four of the fields, the current time and date must only match the non-asterisk field. For example, the following entry specifies that a command should be run at 10 minutes after every hour, on every day of every month.

```
10 * * * *
```

The following example specifies that a command should be executed on the first day of every month at 2:15 in the morning.

```
15 2 1 * *
```

Fields can contain multiple values, either separated by commas (without spaces after the commas) or defined by ranges with a hyphen separating them. For example, the next entry specifies that a command should be executed on the 1<sup>st</sup>, 10<sup>th</sup>, and 20<sup>th</sup> of every month at 1:00 a.m.

```
0 1 1,10,20 * *
```

The time specifications can become very complicated if you need them to be. But by learning the meaning of the five fields and reviewing a few examples, you'll soon be able to construct any time specification you are likely to need.



To learn more about how you can form very complex time specifications for use with the `crontab` command, and to view more examples of time specifications, view the manual page for the `crontab` configuration file by entering the command `man 5 crontab`.

To use the `crontab` command to submit your own cron jobs (as opposed to placing scripts in the `/etc/cron.weekly` subdirectory, for example), you must create a file that has a time specification and a command to execute. Unlike the `at` command, you must create a file containing this information. You cannot enter cron job commands interactively. Suppose for

example, that you wanted to run the `du` command on all home directories and sort the results every morning at 1:00 a.m. You would do this by following these steps:

1. Create a text file containing the time specification followed by the command you want to execute. For this example, the file would contain the following lines (assuming you are entering this cron job as the `root` user):

```
0 1 * * * du | sort > /root/du_results
```

2. Submit the file to the `crond` daemon using the `crontab` command. Assuming that the file you created in Step 1 were called `du_nightly`, the command would look like this:

```
crontab du_nightly
```

You may notice that the file created in Step 1 does not include the `#!/bin/bash` information, as a shell script would. Also, the command shown in Step 1 fits on a single line.

Many times, you may need to do something more complicated with your scheduled administration jobs. Suppose, for example, that you had developed a series of shell scripts that checked the security of the Linux system by reviewing log files, checking network activity, and watching user login activity. If the script that performed all of these activities were called `secure_system`, you could run that script every morning at 1:30 a.m. by adding this line to your `du_nightly` file before submitting the file using the `crontab` command:

```
30 1 * * * /sbin/secure_system
```

The full pathname of the script is included so that the `crond` daemon will be certain to locate the script when it attempts to execute this job. The point of this exercise is that a cron job must be contained on a single line (though it can be a long line with several commands on it). For example:

```
du | sort > file
```

If you want to process more complex cron jobs, create a shell script and execute the script using a cron job.

In the example file `/etc/crontab` shown at the beginning of this section, you saw that some environment variables were defined. You can also include environment variables in the files that you create to define your cron jobs. For example, suppose you want the `du_nightly` cron job to be executed via the `root` user account so that it can access all home directories, but you want an e-mail message sent to your regular user account so it will be waiting in your regular e-mail account. (In other words, you won't have to log in as `root` to access the message.) To do this, you can include the following line in the `du_nightly` file (substituting your own username for `nwells`):

```
MAILTO=nwells
```

You can learn about a few other environment variables supported by the `crontab` command in the `crontab` manual page. You can also use the `-u` option to submit a cron job as

another user. (You can only do this if you are logged in as `root`.) For example, you could submit the `du_nightly` job as user `nwells` using this command:

```
crontab -u nwells du_nightly
```

You can check that the `atd` and `crond` daemons are running by using the `ps` command. For example, to see the `atd` daemon, use this command:

```
ps ax | grep atd
```

To see the `crond` daemon, use this command:

```
ps ax | grep crond
```

These two daemons are started using standard scripts located in `/etc/rc.d/init.d`. You can use those scripts to stop and restart the daemons if needed, but because these daemons carefully check the dates on files to see when new cron jobs have been submitted, you should never need to restart the daemons. The next part of the chapter contains additional information on managing the cron jobs that you have submitted.

---

## MANAGING `at` AND `crontab`

Now that you have learned how to submit commands for future execution using the `at` and `crontab` commands, in this section you learn how to see what commands are scheduled for execution and how to alter those commands when necessary. As with other parts of Linux (such as file permissions and printing), you must be logged in as `root` to work with cron jobs that you have not submitted yourself. The `root` user can submit jobs as any user (using the `-u` option shown previously) and can also view or modify any cron jobs submitted by any user on the system.

### Checking the Status of `at` and `crontab`

All of the commands that you submit using the `at` command or the `crontab` command are stored in a subdirectory of `/var/spool`. Jobs submitted using `at` are stored in the `/var/spool/at` directory; jobs submitted using `crontab` are stored in the `/var/spool/cron` directory. The two types of jobs are stored differently, however. Jobs for the `atd` daemon contain all the environment variables and related information so that the shell can execute the job independent of any other process. Jobs for the `crond` daemon store only a limited amount of information because they are executed in an environment that the `crond` daemon defines for the user who submitted the cron job.

Suppose you enter a single `du` command using the `at` command. When you enter this command, it might appear on screen like this:

```
at now + 10 minutes
at> du
at> <EOT>
```

When you change to the `/var/spool/at` directory, you will see a strange filename such as `a0000d00f08407`. This filename is generated automatically by the `at` command.

Looking inside this file using the `less` command (or a similar command), you will see a number of environment variables. These variables record the facts about the shell in which the command was entered via the `at` command, so that the same shell environment can be duplicated when the command is executed. The last line of the file is the command (in this example, `du`) that will actually be executed at the specified time. An example file from the `/var/spool/at` directory is shown here. The values in a file from your system will vary.

```
#!/bin/sh
# atrun uid=0 gid=0
# mail      root 0
umask 22
USERNAME=root; export USERNAME
ENV=/root/.bashrc; export ENV
HISTSIZE=1000; export HISTSIZE
HOSTNAME=localhost.localdomain; export HOSTNAME
LOGNAME=root; export LOGNAME
HISTFILESIZE=1000; export HISTFILESIZE
MAIL=/var/spool/mail/root; export MAIL
HOSTTYPE=i386; export HOSTTYPE
PATH=/usr/bin:/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin:/root/
bin; export PATH
KDEDIR=/usr; export KDEDIR
HOME=/root; export HOME
INPUTRC=/etc/inputrc; export INPUTRC
PS1=[\u@\h\ \W]\$ ; export PS1
USER=root; export USER
OSTYPE=Linux; export OSTYPE
SHLVL=1; export SHLVL
cd /root || {
    echo 'Execution directory inaccessible' >&2
    exit 1
}
du | sort > /tmp/du_listing
```

The information in the `/var/spool/at` directory does not, however, indicate the time when the command will be executed. Because of the complicated format of these files and their interaction with the `atd` daemon, you should not directly modify the `/var/spool/at` directory using a command such as `rm`. Instead, use the `atq` and `atrm` commands to view jobs submitted using `at` that are pending execution.

The `atq` command lists each of the jobs processed by the `at` command. For each job, this command lists a job number and the date and time when the job will be executed. To use the `atq` command, simply enter it at any shell prompt. No parameters are used. Sample output from the `atq` command is shown here:

```
6      2000-08-15 10:00 a
9      2000-12-21 05:15 a
12     2000-12-31 11:45 p
```

You can also use the `at` command with the `-l` option (for *list*) to see a list of jobs pending execution by `atd`. This command looks like this:

```
at -l
```

If you decide to cancel a command that you have submitted using `at`, use the `atrm` command. The `atrm` command deletes (removes) a job from the queue that `atd` uses to execute commands. (A **queue** is a list of commands or files to be processed.) You can also use the `-d` option (delete) with the `at` command to remove a job from this queue, so that it is not executed. Both `atrm` and `at -d` require you to include the job number, which you must obtain from the `atq` (or `at -l`) command. For example, to remove job 11, you could use either of these commands:

```
atrm 11
at -d 11
```

You can use a similar set of commands to manage jobs that have been submitted using `crontab`. To begin, you'll notice that the file stored in `/var/spool/cron` is different from the files stored in `/var/spool/at`. The `/var/spool/cron` directory contains a single file for each user who has submitted jobs using `crontab`. For example, if you have submitted a job using `crontab` while logged in as `root`, the file `/var/spool/cron/root` exists. If you submitted a job using `crontab` while logged in as `nwells`, the file `/var/spool/cron/nwells` exists. The file in `/var/spool/cron` that is named for your username contains a composite of all cron jobs that you have submitted. For example, when you have submitted a single cron job, the file looks something like this:

```
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (secure_cron installed on Mon Oct 20 18:12:16 2000)
# (Cron version -- $Id: crontab.c,v 2.13 1994/01/17 03:20:37 vixie
Exp $)
30 1 * * * /sbin/security_scan
```

As with the contents of the `/var/spool/at` directory, you should not directly edit a `crontab` file in `/var/spool/cron`. Instead, use the options provided by the `crontab` command. These are summarized in the following list:

- `crontab -l` lists the contents of your `crontab` file (for the user account you are currently logged in to).
- `crontab -r` removes your `crontab` file. Use this option carefully, as it removes the entire file; so any cron jobs you have submitted are lost. Remember that you can store system administration scripts in the subdirectories such as `/etc/cron.daily` and `/etc/cron.weekly` to have them executed at regular intervals without creating a separate `crontab` file for the `root` user account.
- `crontab -e` opens your `crontab` file in a text editor, so you can make changes in the times or commands defined for your cron jobs. It's important that you use the `-e` option on the `crontab` command rather than using a regular editor session (such as `vi /var/spool/cron/nwells`) to change your `crontab` file. Using `crontab -e` prevents file locking conflicts that would cause problems with the `crond` daemon.

## Controlling Access to `at` and `crontab`

The default settings on most Linux systems allow any user to submit commands for future execution using either `at` or `crontab`. You can create a set of configuration files that restrict access to the `at` and `crontab` commands so that only those users specified by the system administrator can use these commands. The files that enforce this control are as follows:

- `/etc/cron.allow`: contains usernames (one per line) that are allowed to use the `crontab` command
- `/etc/cron.deny`: contains usernames (one per line) that are not allowed to use the `crontab` command
- `/etc/at.allow`: contains usernames (one per line) that are allowed to use the `at` command
- `/etc/at.deny`: contains usernames (one per line) that are not allowed to use the `at` command



On some Linux systems, the files listed above may be located in the `/var/spool/cron` and `/var/spool/at` subdirectories, in which case they will be named simply `allow` and `deny`, without the name `at` or `cron` as a prefix.

On most Linux systems, none of the four files just listed exist, meaning that any user can use both `at` and `crontab`. (On some systems, however, having none of these files may mean that only `root` can use `at` and `crontab`.) When you attempt to use the `at` or `crontab` command, the command checks the permission files in the following order:

- If the `/etc/cron.allow` file exists, a user must be listed in that file in order to use the `crontab` command. The same rule applies to `/etc/at.allow` for the `at` command.
- If the `cron.allow` (or `at.allow`) file does not exist, but the `cron.deny` file does exist, any user listed in `cron.deny` (or `at.deny`) cannot use the `crontab` command (or the `at` command).

By controlling access to the `at` and `crontab` commands on a busy Linux system, you can make it more difficult for regular users to consume system resources when you are completing (or scheduling for automatic execution) routine system administration tasks. In addition, users who have access to the `at` and `crontab` commands can create security hazards by automatically checking for and perhaps refreshing security loopholes that those users take advantage of on an insecure system.

## CHAPTER SUMMARY

- Computer programs can be written using a compiled or an interpreted programming language. Each type has its own advantages. Shell scripts are interpreted programs, with the shell acting as the command interpreter. Using shell scripts is a powerful way to

automate many tasks on a Linux system. Shell scripts include commands that you would otherwise execute from a command line, as well as commands that are normally used only within script files, such as selection statements (`if/then/else`) and loop statements (`for` and `while`). Shell scripts can be executed by a variety of shells. Other types of scripting languages such as Perl, Tcl/Tk, Python, and Expect are also used for automating system administration tasks. All standard Linux systems have the capability to use these standard types of scripts.

- The `at` and `crontab` commands are used to automate one-time and repetitive tasks respectively. Both `at` and `crontab` permit you to use complex time specifications (though the two commands use different formats). Commands submitted for future execution using the `at` command can be entered interactively or using a file containing the commands. Commands to be executed using `crontab` must be placed in a file that also contains the execution time for those commands.
- Additional commands let you see what cron jobs are currently awaiting execution by either the `atd` daemon or the `crond` daemon. Both are normally running on Linux systems. The `atq` and `atrm` commands help you manage jobs submitted using `at`. The `-r`, `-l`, and `-e` options on the `crontab` command provide similar functionality for jobs submitted using `crontab`. Configuration files such as `/etc/cron.allow` let you control how regular users access the `at` and `crontab` commands.

---

## KEY TERMS

**\$@** — A special shell variable that includes all of the parameters on the command line.

**at** — A command that lets you enter one or more commands to be executed once at some future time.

**atq** — A command that lists each of the jobs that have been submitted using the `at` command, with a job number and the date and time when the job will be executed.

**atrm** — A command that deletes (removes) a job from the queue used by `atd` to execute commands.

**batch** — A command that executes scheduled tasks when the system load average drops below 0.8.

**binary file** — See executable file.

**comment** — A line in a script that begins with a `#` character. Comments are not processed by the shell, but are only included to help someone reading the file understand the purpose of the script or how it functions.

**Common Gateway Interface (CGI)** — A method of communication between two programs using the standard input and standard output channels.

**compiled language** — A computer language for which the source code is converted to a binary file long before the program will be run by users.

**compiler** — A special program that converts the source code of a compiled language into a binary file.

**computer language** — A set of words and syntax rules that can be arranged in predefined ways to cause a computer to perform tasks defined by the person using the language.

- cron job** — A command or script that you have scheduled to be executed at a specific time in the future.
- crontab** — A command that lets you enter one or more commands to be executed repeatedly at intervals that you designate.
- do** — A command used with the **done** command to enclose a block of commands to be executed by a **for** command.
- done** — A command used with the **do** command to enclose a block of commands to be executed by a **for** command.
- echo** — A command that prints text to the STDOUT channel—to the screen unless output has been redirected.
- else** — A command that extends the capability of an **if/then** statement by adding a block of commands that are only executed if a test returns a value of false (that is, if the test fails).
- executable file** — A file containing numeric codes that a computer can execute. Created from a source code file by a compiler, the executable file is the program that a user can run.
- exit** — A command that stops the execution of a script immediately.
- fi** — A command that marks the end of an **if/then** statement.
- for** — A command that repeats a block of commands one time for each item in a list that you provide.
- for loop** — A block of commands that is repeatedly executed according to the parameters provided with the **for** command.
- if** — A command used to introduce a test within a shell script. An **if** command is always followed by a **then** command.
- if/then/else statement** — A set of commands used to determine whether other commands in a script are executed. An **if/then/else** statement is one kind of selection statement.
- index** — A counting variable used within a loop statement. The index acts as a marker to count how many times the loop has executed a list of commands.
- interpreted language** — A language for which the source code is converted to numeric codes at the time a user runs the program. This conversion takes place each time the user runs the program. For this reason, interpreted languages are generally much slower than compiled languages.
- interpreted program** — A computer program that is converted from human-readable form to a format that can be used by a computer (numeric codes) at the moment you execute the program.
- interpreter** — A special program that converts the source code of an interpreted language into numeric codes that a computer can execute.
- iteration** — An occurrence of an event or process that can or must be done many times.
- keywords** — Words used in a computer language to define a specific task or meaning.
- loop statement** — A statement used to determine whether part of a computer program should be executed more than once.

- make** — A command that uses information stored in a configuration file to compile source code into a binary program, or otherwise prepare a program to be executed.
- nesting** — A programming method in which one selection or loop statement contains another selection or loop statement.
- Perl** — A programming language developed by Larry Wall initially to process strings of text and generate reports.
- positional variable** — A variable used within a shell script that contains data included on the command line when the script was launched.
- programming language** — *See* computer language.
- Python** — A scripting language developed by Guido van Rossum that is often used for creating graphical programs.
- queue** — A list of commands or files to be processed.
- script** — A text file that can be interpreted or executed by another program (an interpreter).
- selection statement** — A statement that lets a computer programmer determine which parts of a program will be executed according to values that are calculated by testing as the program is executed. The `if/then` statement is a selection statement used in shell scripts.
- shell script** — An executable file containing lines of text as you would enter them at a command line, including special commands to control the order in which lines in the file are executed.
- shell variable** — A variable used within a shell script to store information for use by the script.
- source code** — The file that a programmer writes using the keywords and syntax rules of a computer language.
- statement** — A command within a computer program. A statement is often a single keyword, but the term may also refer to a group of keywords that the computer language syntax requires or allows to be used together.
- statement block** — A list of commands (or statements) that are controlled by a selection or loop statement.
- Tcl/Tk** — A scripting language developed by John Ousterhout that is used to create graphical programs.
- test** — A command that evaluates the arguments provided after the command name and returns either true (a value of 1) or false (a value of 0).
- test** — A method of examining data within a shell script and acting according to the result of the examination (or test).
- then** — A command that identifies the commands to be executed if the test introduced by the `if` command succeeds (returns a value of true).
- tty** — A command that displays the name of the terminal device you are currently working in.
- while** — A command that creates a loop based on a test. The loop executes a block of commands as long as the test returns true.
- white space** — Tabs or spaces included in a program or script that make the script easier for a person to read.

## REVIEW QUESTIONS

1. A shell script is similar to a DOS batch file in that both are used to automatically execute commands that would otherwise be entered manually. True or False?
2. Name four shell scripts that are included on a standard Linux system, and describe the use of each one.
3. The first line of a standard shell script must contain:
  - a. A comment defining the username of the person creating the script
  - b. A valid command as you would enter it from a command line
  - c. The path and filename of the shell used to execute the script
  - d. A time specification for when the script will be executed
4. In order to be executed by any user (including `root`), a shell script must have the file permission set.
  - a. Execute
  - b. Write
  - c. Other
  - d. Owner
5. When you are working in the `bash` shell, all shell scripts that you execute must use `bash` shell syntax. True or False?
6. A shell variable is one that is:
  - a. Restricted for use only by the shell itself
  - b. Available to all parent processes
  - c. Used to hold a value within a shell script
  - d. Stored in virtual memory
7. Which of these statements contains a standard positional variable?
  - a. `gzip $file`
  - b. `gzip $1`
  - c. `gzip /tmp/listing 2>&1`
  - d. `gzip HOME`
8. Which two commands involve a test value of true or false?
  - a. `if`
  - b. `while`
  - c. `for`
  - d. `crontab`
9. The `test` command is equivalent to using square brackets around a test expression, but the `test` command is less frequently used. True or False?

10. Briefly explain why comments are an important part of any shell script.
11. Describe the difference in control methods between a `for` loop and a `while` loop.
12. A loop beginning with the command "`for i in 2 4 6 8`" will be executed how many times?
  - a. Eight
  - b. Four
  - c. The preset value of `i`
  - d. It cannot be determined without knowing what files are in the working directory.
13. Name three non-shell scripting languages. Include a statement on the use, characteristics, or author of each one.
14. CGI is popular for which of the following purposes?
  - a. Creating Web servers
  - b. Interfacing between scripts and Web servers
  - c. Automating system administration work
  - d. Creating graphical programs
15. Briefly explain the difference in time specification formats for the `at` and `crontab` commands.
16. The `crontab` and `at` commands both support an interactive mode in which you can enter commands to be scheduled for future execution at the command line. True or False?
17. The daemon(s) that manage(s) commands submitted using the `at` and `cron` commands is (are):
  - a. `/etc/cron.allow`
  - b. `crond`
  - c. `atd` and `crond`
  - d. `init.d`
18. When using the `at` command interactively to enter commands scheduled for future execution, you indicate that you have finished entering commands by pressing:
  - a. Ctrl+D
  - b. Ctrl+C
  - c. Esc
  - d. Ctrl+X
19. The `batch` command is used to schedule jobs so that:
  - a. Jobs from the `at` and `crontab` commands are not executed at the same time.
  - b. Commands used in a DOS environment can be executed by Linux.
  - c. The system will not be overloaded with scheduled tasks.
  - d. Regular users can schedule tasks.

20. A simple method for `root` to schedule recurring system administration tasks is to:
  - a. Use the `at` command in interactive mode.
  - b. Add a script to a directory such as `/etc/cron.daily` or `/etc/cron.weekly`.
  - c. Create a graphical program using Python with Tk extensions.
  - d. Debug existing shell scripts on the system.
21. The output of a cron job is normally sent via \_\_\_\_\_ to the user who submitted the cron job or to the user defined by the \_\_\_\_\_ variable within the file containing the cron job.
  - a. e-mail, `MAILTO`
  - b. e-mail, `USERLOG`
  - c. tty, `MAILTO`
  - d. standard output, `USERLOG`
22. Name in order from left to right the fields of the `crontab` time specification, giving the range of valid values for each one.
23. Output from a cron job cannot be redirected using standard redirection operators because the environment in which the cron job was created is unlikely to exist when the cron job is executed. True or False?
24. The \_\_\_\_\_ file can include a username in order to deny that user access to the `crontab` command.
  - a. `/etc/cron.allow`
  - b. `/etc/cron.deny`
  - c. `/usr/local/bin/deny`
  - d. `/etc/at.deny`
25. Name the two separate commands that can be used instead of `at -l` and `at -d`.

## HANDS-ON PROJECTS



### Project 12-1

In this activity you use the `vi` editor to create a simple shell script. Then you execute that script at the command line. To complete this activity you should have a working Linux installation with a valid user account.

1. Log in to Linux using your username and password.
2. If you are using a graphical environment, open a terminal window so you have a shell prompt.
3. Enter `vi fileinfo` to start `vi` and create a new file called `fileinfo`.
4. Press `i` to start insert mode in `vi`.

5. Enter the following lines in `vi`:

```
#!/bin/bash
if [ $# -lt 1 ]; then
    echo You must include a filename on the command line.
    exit
fi
echo Beginning to process files.
for i in $@
do
    echo Number of lines in $i
    wc -l $i
    echo Number of words in $i
    wc -w $i
done
```

6. Press **Esc** to return to command mode.
7. Press **Shift+Z**, **Z** to exit `vi` and save the file.
8. Enter `chmod 755 fileinfo` to change the file permissions on the file you created to include the execute permission. What would be the equivalent command using letters instead of the 755 code?
9. Enter `./fileinfo` to execute the command without any parameters. What happens? What result is returned by the test in the second line of the script? What other methods could you use to test for the presence of a command-line parameter? What would happen if you removed the `if/then` test but didn't include any filenames on the command line?
10. Enter `./fileinfo /etc/termcap` to execute the command with a parameter. What happens? How many times was the `for` loop executed?
11. Execute the command with the parameter shown here: `./fileinfo /etc/c*conf`. What happens? Do you see any error messages? Why? How could you alter the script to prevent those error messages from appearing on screen? How could you alter the script to test the validity of each filename before using the `wc` command to prevent the error code from being generated in the first place? (*Hint*: Use the `-r` file test from Table 12-1.)



## Project 12-2

In this activity you submit a job for future execution using the `at` command. To complete this activity you should have a working Linux installation with a valid user account.

1. Log in to Linux using your username and password.
2. If you are using a graphical environment, open a terminal window so you have a shell prompt available.
3. Enter the `at` command with a time designation as follows: `at now + 15 minutes`. The `at>` prompt appears.
4. Enter the loop command `for i in /etc/*conf` at the first `at>` prompt.

5. Enter the **do** command at the next **at>** prompt.
6. Enter **wc -w \$i** at the next **at>** prompt.
7. Enter **done** at the next **at>** prompt. Your screen should now look like this:

```
$ at now + 15 minutes
at> for i in /etc/*conf
at> do
at> wc -w $i
at> done
at>
```

8. Press **Ctrl+D** to finish entering the commands you want to automate. What message do you see? What time is specified? Is it a relative time or an absolute time based on the time specification that you entered?
9. Enter the **atq** command. What information do you see about the job you just entered? What is the job number? Where could you find the job information stored in the file system?
10. If you wish, use the **atrm** command to remove the job from the **at** queue, using the job number given by the **atq** command. For example, you might use the command **atrm 15**.
11. If you decided not to remove the **at** job you entered, what will happen at the time given by the **at** command (when you finished the entry with **Ctrl+D**)? Wait 15 minutes and then use a mail reader (such as mail, elm, pine, or Netscape) to view the output of your **at** job as an e-mail message.



## Project 12-3

In this activity you submit a job for future execution using the **crontab** command. You then switch to the **root** account and modify a user's access to **crontab**. To complete this activity you should have a working Linux installation with a valid user account and access to the **root** account on the system. (If you don't have **root** access, you can complete all but the last few steps of the project.)

1. Log in to Linux using your regular username and password.
2. If you are using a graphical environment, open a terminal window so you have a shell prompt available.
3. Enter **vi du\_job** to start the **vi** editor with a new filename of **du\_job**.
4. Press **i** so that you can enter text in insert mode.
5. Enter **30 2 \* \* \* du /home > /tmp/du\_output** in **vi**.
6. Press **Esc** to return to command mode.
7. Press **Shift+Z**, **Z** to save the text you entered and exit **vi**. Describe the time specification that you entered in the **du\_job** file. What problem could occur with the command entered in the **du\_job** file because you are working as a regular user?
8. Enter **crontab du\_job** to submit the file you created as a new cron job.

9. Enter **crontab -l** to review the contents of the **crontab** file for your regular user account. What do you see? How does it relate to the information you entered in the **du\_job** file? Can you see an indication of the filename **du\_job** and when it was submitted using **crontab**?
10. Enter **su** to change to the **root** account.
11. Enter the **root** password when requested.
12. Enter **vi /etc/cron.deny** to create a new file called **cron.deny** in the **/etc** directory.
13. Enter a single line in the **cron.deny** file that includes the regular user account name that you used to log in, in Step 1. For example, the file might include a single line like this: **nwells**.
14. Press **Shift+Z, Z** to save the text you entered and exit **vi**.
15. Enter **exit** to switch from the **root** account to a regular user account again.
16. Enter **crontab -r** to remove the **crontab** file for your regular user account. What happens? Why?
17. Enter **su** to change back to the **root** account, entering the **root** password when prompted.
18. Enter a command such as **crontab -u nwells -r** to erase the **crontab** file for the regular user account. (Substitute your own regular user account name for **nwells** in this example.)
19. Enter **rm /etc/cron.deny** to remove the **cron.deny** file so that in the future all user accounts will have access to the **crontab** command (unless you prefer to restrict it for some reason on this Linux system).
20. Enter **exit** to log out of the **root** account.

---

## CASE PROJECTS

1. You are working as a senior system administrator at Image Makers, a large advertising agency and public relations firm in New York City. The office maintains a large Web site that contains advertising resources and samples of the agency's work for potential clients to review. The office also includes several servers and nearly 100 workstations that rely on those servers for data storage and Internet connectivity. Most of the servers are running on Linux. The desktop systems are Windows 2000 and Macintosh. The agency's management is concerned that the firm's Internet site is a potential victim of defacement by hackers because of some advertising the company has done for clients in the computer security business. Based on what you have learned so far about system administration and the tools described in this chapter, describe some of the most important system administration tasks you would like to automate using the **crontab** command. (Backing up the system would be one of these tasks, but that is the subject of a future chapter.)

2. Some employees at the agency want to use special Web browser programs to automatically retrieve large Web pages during the night. Are these types of programs compatible with the `crontab` command? Would you prefer that employees use the `crontab` command on the server? Why or why not? What issues would this bring up related to (a) security, (b) system load, and (c) end-user training?
3. After six months of continuous work at Image Makers, you finally have a day off scheduled for next Friday. During the four days before your day off, you come across several tasks that require action on Friday. You cannot do them before you leave because of the system load during work hours. Also, much of the data needed for the Friday reports will not be available until the end of the day on Friday. Describe the commands you would use to set up these one-time tasks for the time you are away from the office. Would you choose the `at`, `batch`, or `crontab` command? Why? How well do you need to know the company's systems and the Linux commands in order to perform this type of system administration feat?

